Group commit and related enhancements to the MariaDB binary log

Michael “Monty” Widenius
monty@askmonty.org
Kristian Nielsen
knielsen@askmonty.org

Monty Program Ab

O’Reilly MySQL Conference & Expo 2011

Outline

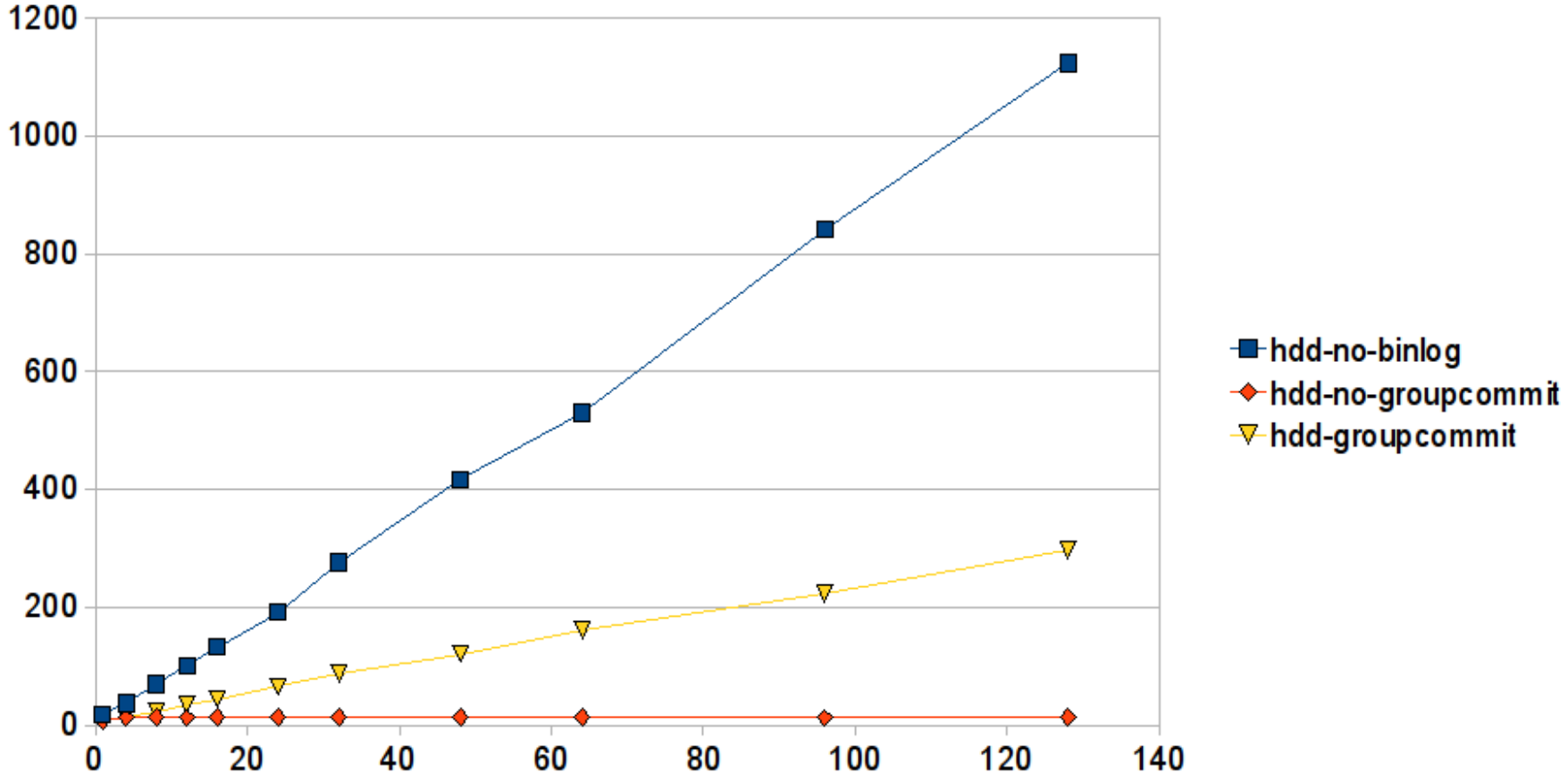
Outline

Your choice: poor performance or no crash recovery

- InnoDB/XtraDB + binlog need **3 fsync()s** for every commit!
- `fsync()` is expensive (BBU raid) to horrendously expensive (HDD).
- Can improve by setting `innodb_flush_log_at_trx_commit=0/2` and `sync_binlog=0`
  - Replication will be hosed after master crashes
  - No durability (can loose commits during crash)
- “A choice between two evils”

`sync_binlog=1` performs badly

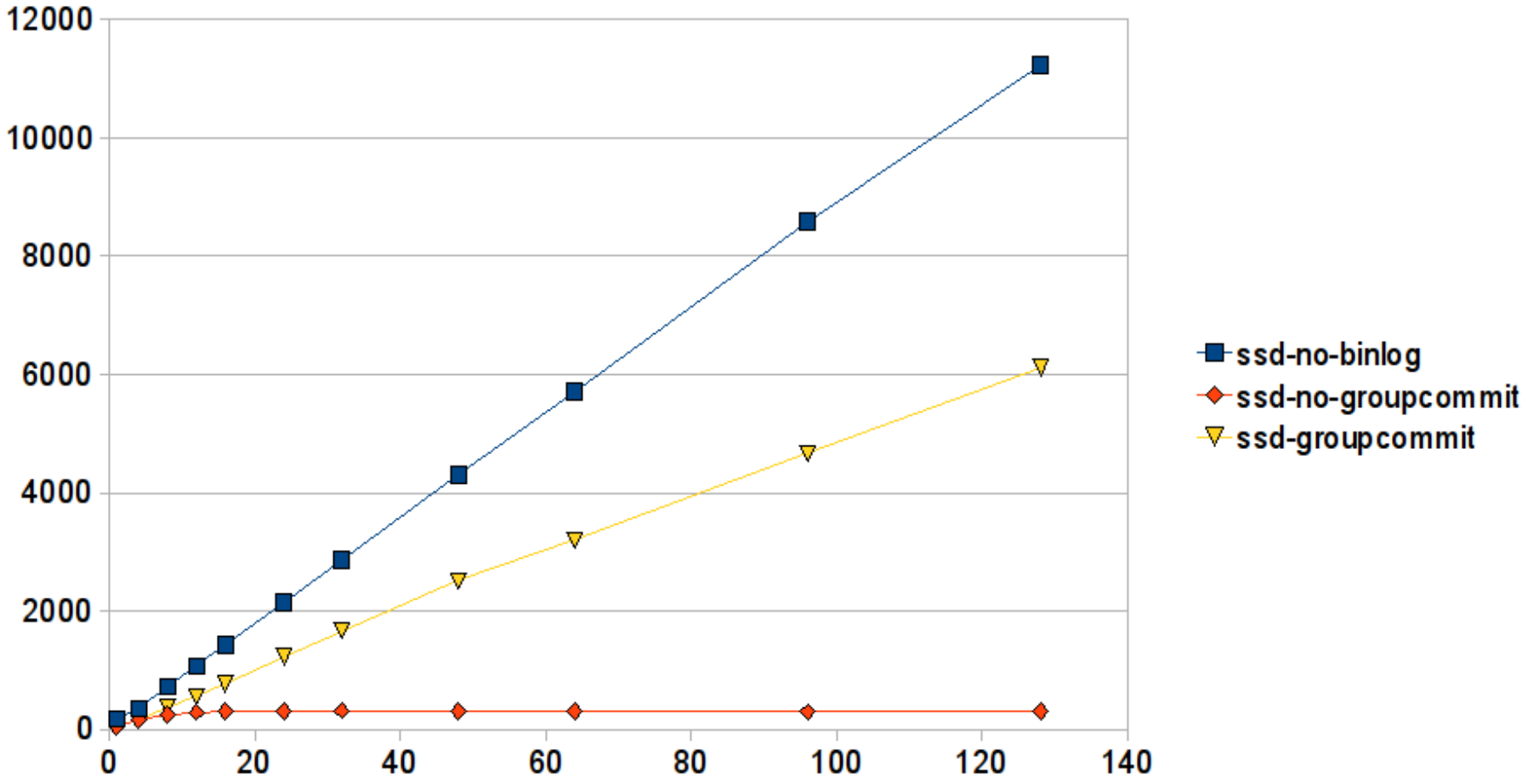
Commits per second vs. number of connections, RAID 1 HDD



- Blue line is with binlog disabled
- Red line shows extreme penalty of enabling binlog

SSD does not help enough. . .

Commits per second, RAID 1 SSD with controller cache enabled



Same picture, enabling binlog (red) has extreme penalty compared to disabled binlog (blue)

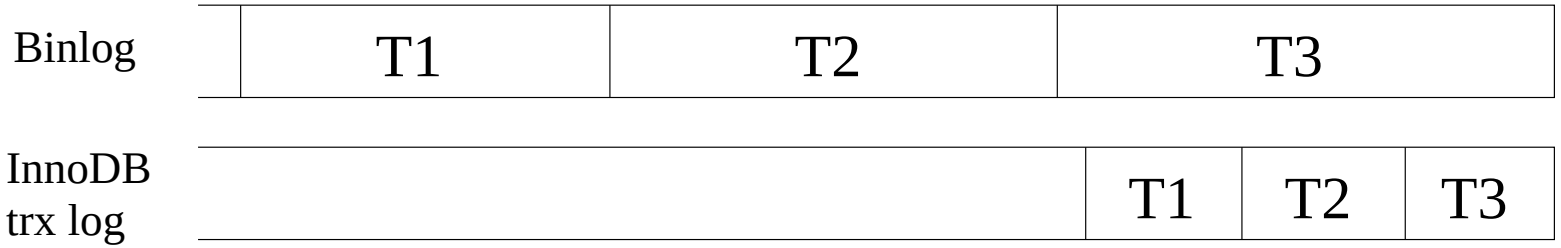
Benchmark

- `innodb_flush_log_at_trx_commit=1`, and `sync_binlog=1`
- Simple transactions
  `REPLACE INTO t(a,b) VALUES ...`
- 1M rows (fits in memory)
- Intel 12-core (24 hyperthread) server with 24GByte RAM
- RAID 1 HDD / Intel SSD
- Plot commits per second versus number of parallel threads
- Benchmark heavily bottlenecked on `fsync()` I/O
- No scaling
- We need to fix this!

Benchmark

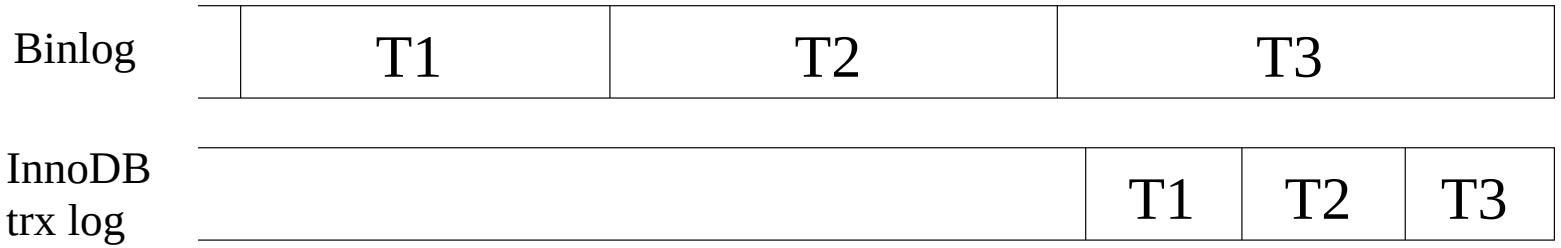
- MySQL Bug#13669
- Filed **30 September 2005** by Peter Zaitsev
- About time that this was fixed

Multiple transaction logs



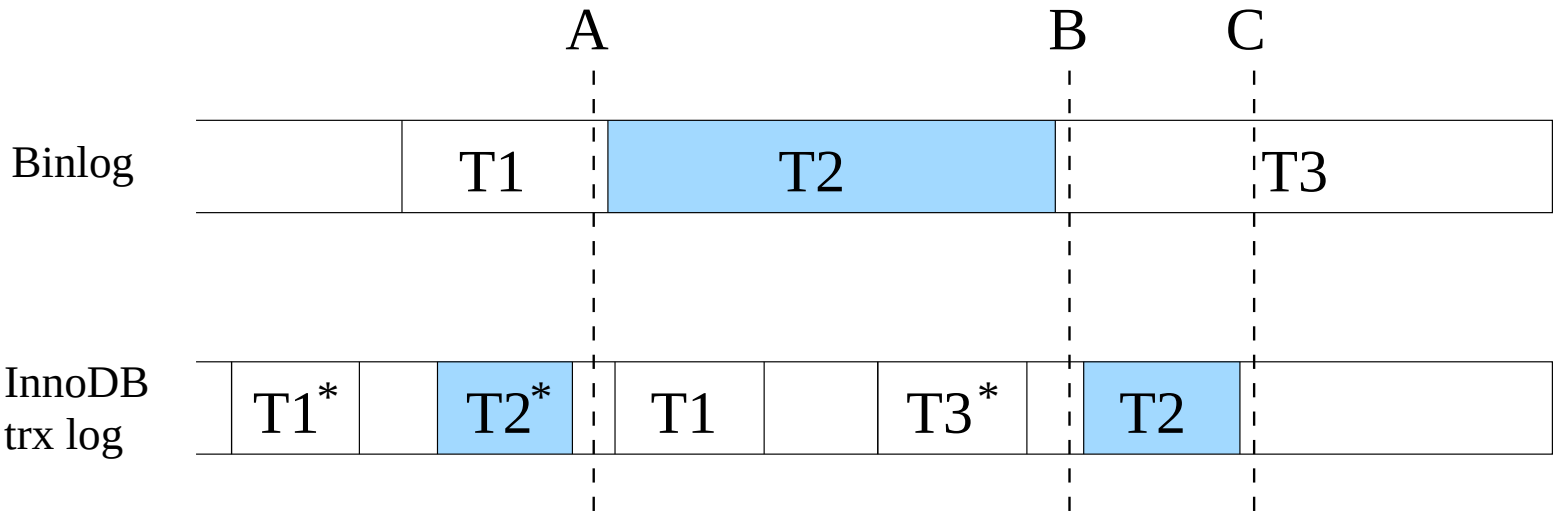
- InnoDB transaction log records commits in order
  - When trx hits the InnoDB log, it is committed
  - Tablespace pages updated later, asynchroneously
  - Pluggable storage engines
- Binlog records transactions in binlog order
  - Transactions in the binlog are executed on slaves during replication
  - When trx hits the binlog, it can be applied on a slave

Multiple transaction logs



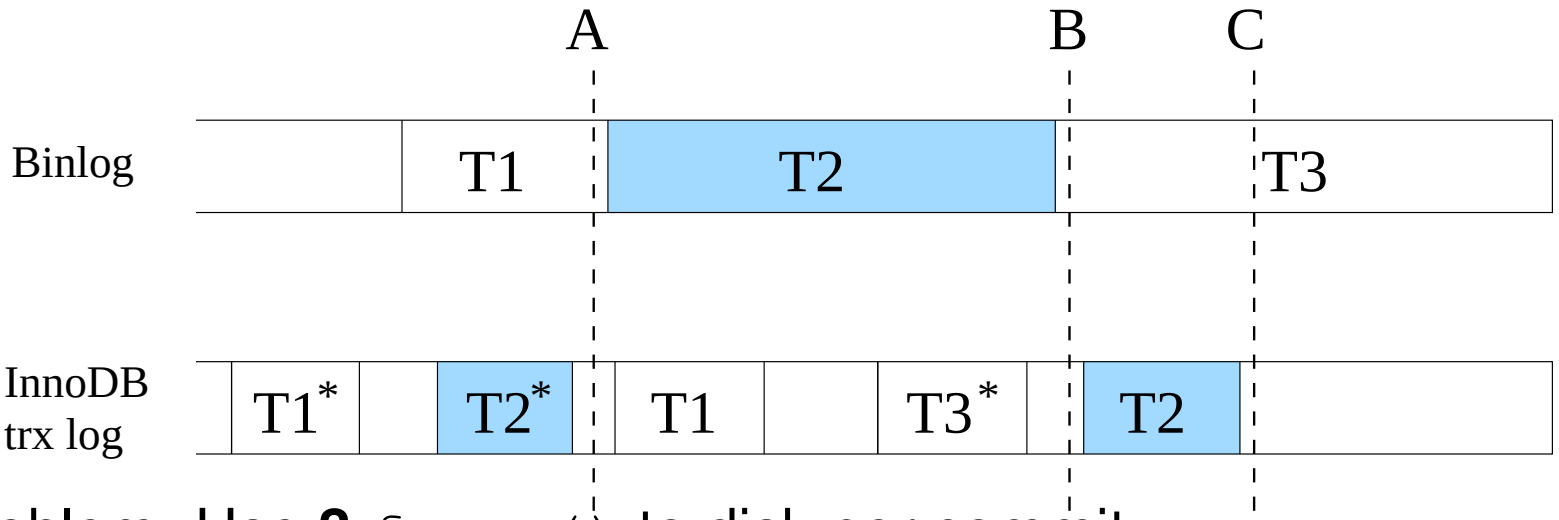
- **Multiple** transaction logs
- After crash, must ensure that all logs have the same transactions committed
  - InnoDB trx log decides what data is on the master
  - Binlog decides what data is on the slave
  - Replication can diverge if they are inconsistent with each other
  - Inconsistency requires full restore of master from backup or re-initialising all slaves
- Done using standard 2-phase commit

2-phase commit



- First we **prepare** (∗) T2 in InnoDB
- Then we **commit** T2 in binlog
- Finally we **commit** T2 in InnoDB
- After crash we will rollback (A), commit (B), or do nothing (C) in InnoDB
- Binlog is authoritative on what is committed and what is not
- Guarantees both **consistency** and **durability**

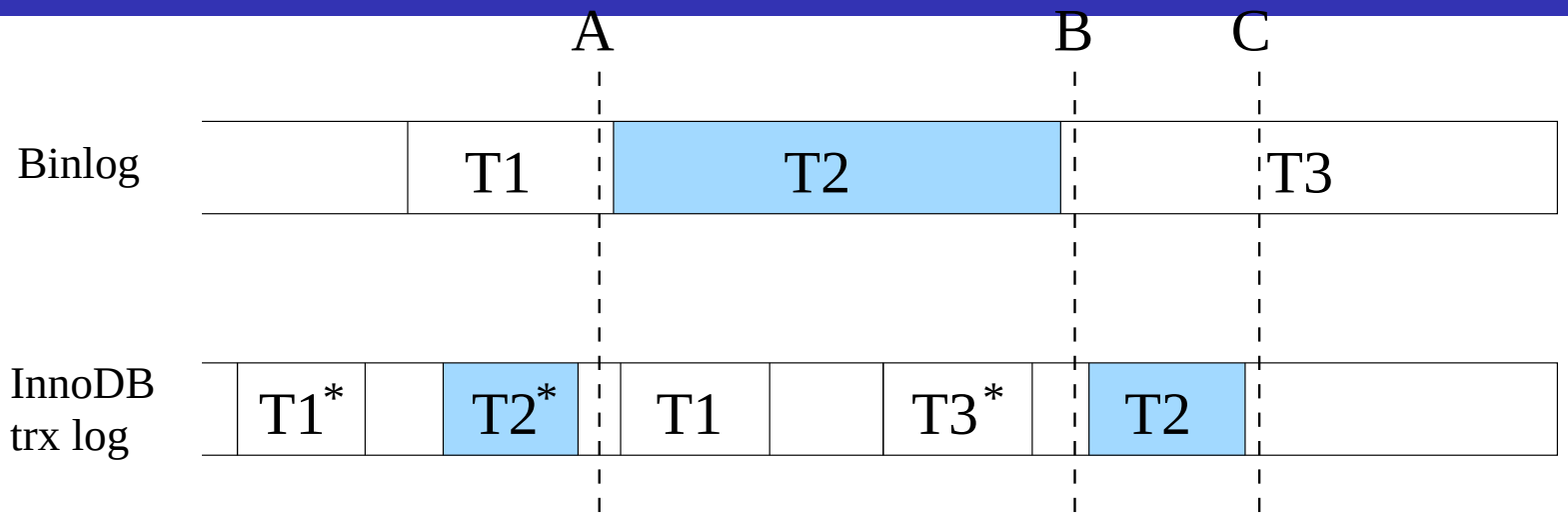
Problem: `fsync()` is expensive



Problem: Use **3** `fsync()` to disk per commit

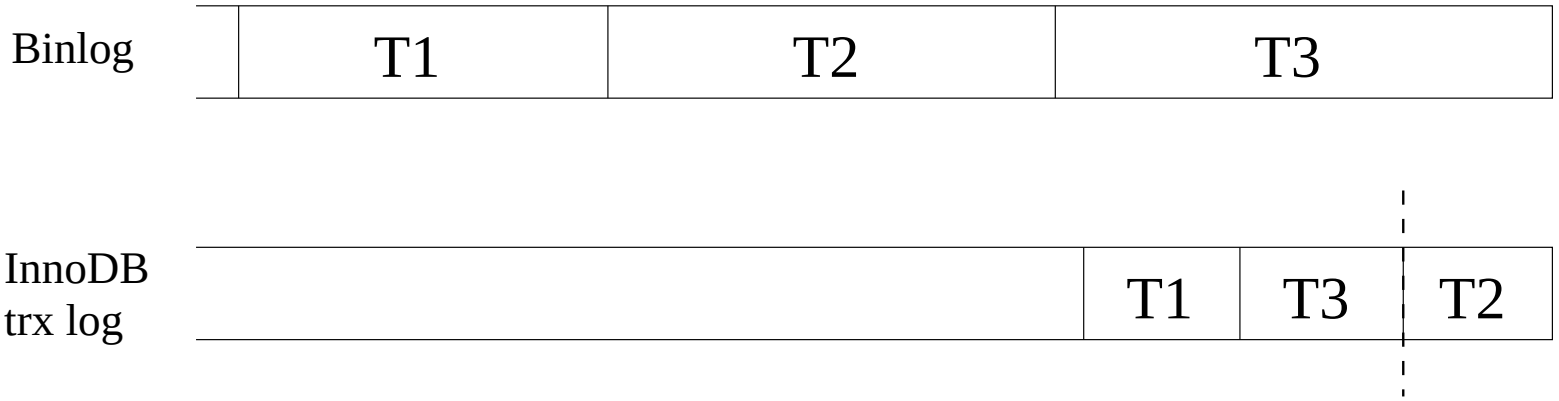
- (A), or can write T2 in binlog to disk before T2* prepare in InnoDB trx log
  - After crash unable to recover T2 in InnoDB
- (B), or can write T2 commit in InnoDB before T2 in binlog
  - After crash unable to roll back T2 in InnoDB
- `fsync()` (C), to know where to start crash recovery
  - Cannot keep binlogs forever

Problem: `fsync()` is expensive



- `fsync()` is expensive
  - Especially on traditional commodity hard disks (ca. 10 msec)
  - Also SSD (in our test around 3 msec) or even with battery-backed-up RAID
  - (10 msec corresponds to around 10,000,000 instructions)
- Standard solution: group commit
  - Write and `fsync()` many parallel transactions at once
  - Amortise the cost of `fsync()` over many commits.

Consistent commit order



Need same commit order in different engines and in binlog

- Online backup takes snapshot of engine
- Could end with engine state that does not exist in binlog
- Unable to provision a slave from above snapshot
  - Will either miss T2, or duplicate T3
- This is reason InnoDB currently serialises all commits, breaking group commit and hurting performance
- Need a better solution

SMP considerations

- Ensuring a particular commit order requires serialisation
  - One commit at a time
- Need care to not cause bad performance on multi-core SMP
- Avoid long queue of threads waiting one after the other
  - Context switches are not free
  - Ties the hands of the kernel thread scheduler
  - If the core is busy that last ran the next-in-line thread, need either expensive migration to other core, or have all following threads wait
  - Best to run the serial part in a single thread

Outline

Extend the storage engine API

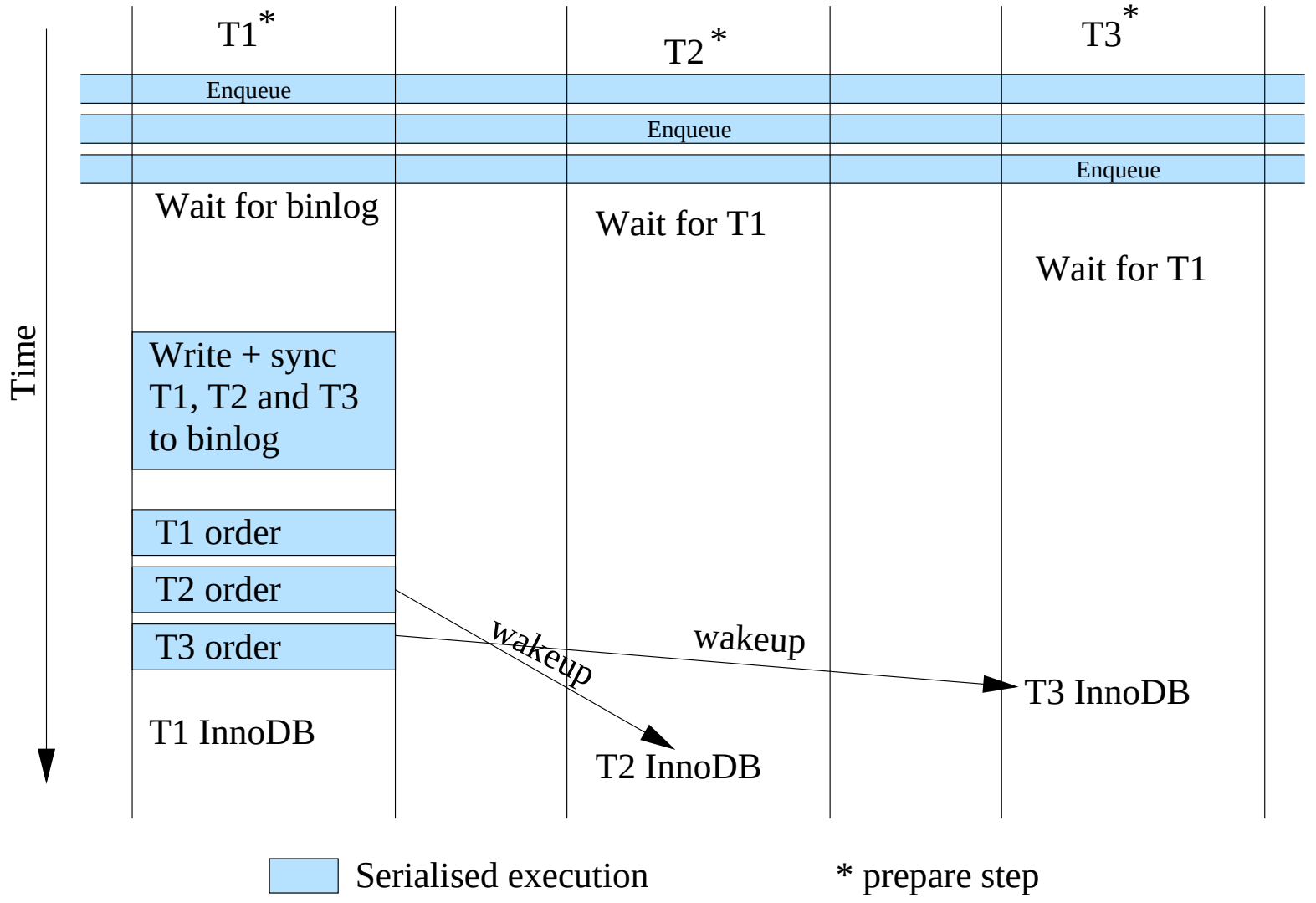
- Split handlerton `prepare()` and `commit()` methods
  - Fast, serialised part that operates in-memory (optional)
  - Slow, parallel part that does I/O
- Implemented in XtraDB and PBXT
  - Also easy to implement in Aria
- Small change to storage engines (few 100 lines)
- Non-supporting storage engines will have group commit but not consistent commit order

Group commit algorithm

1. Do slow part of `prepare()` in parallel in InnoDB (first `fsync()`, InnoDB group commit)
2. Put transaction in queue, deciding commit order
3. First in queue runs the serial part for all, rest wait
   1. Wait for access to the binlog
   2. Write all transactions into binlog, in order, then sync (second `fsync()`)
   3. Run the fast part of `commit()` for all transactions, in order
4. Finally, run the slow part of `commit()` in parallel, (third `fsync()`, InnoDB group commit

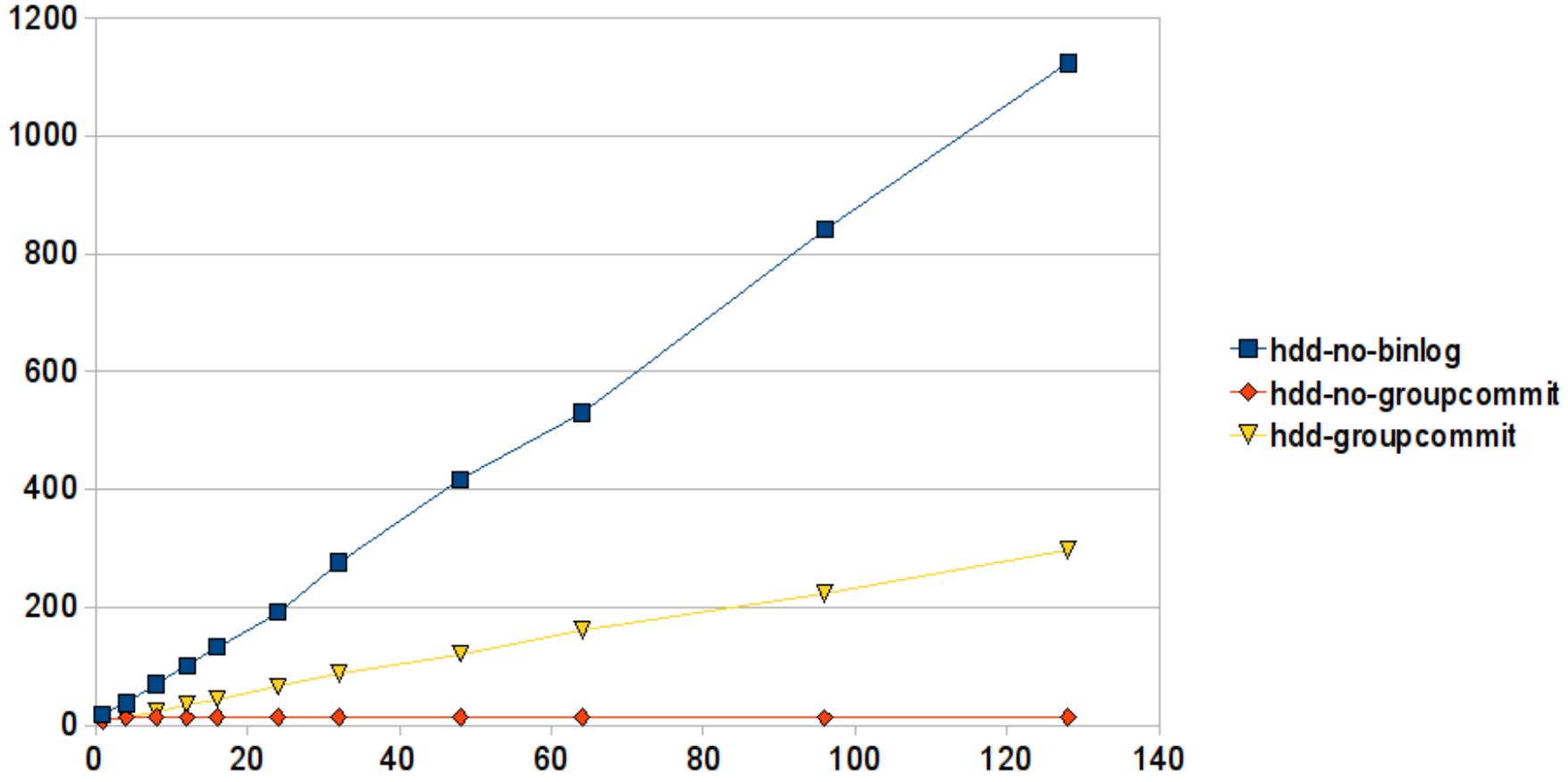
Only two context switches per thread (one sleep, one wakeup)

The new commit algorithm



Group commit scales well

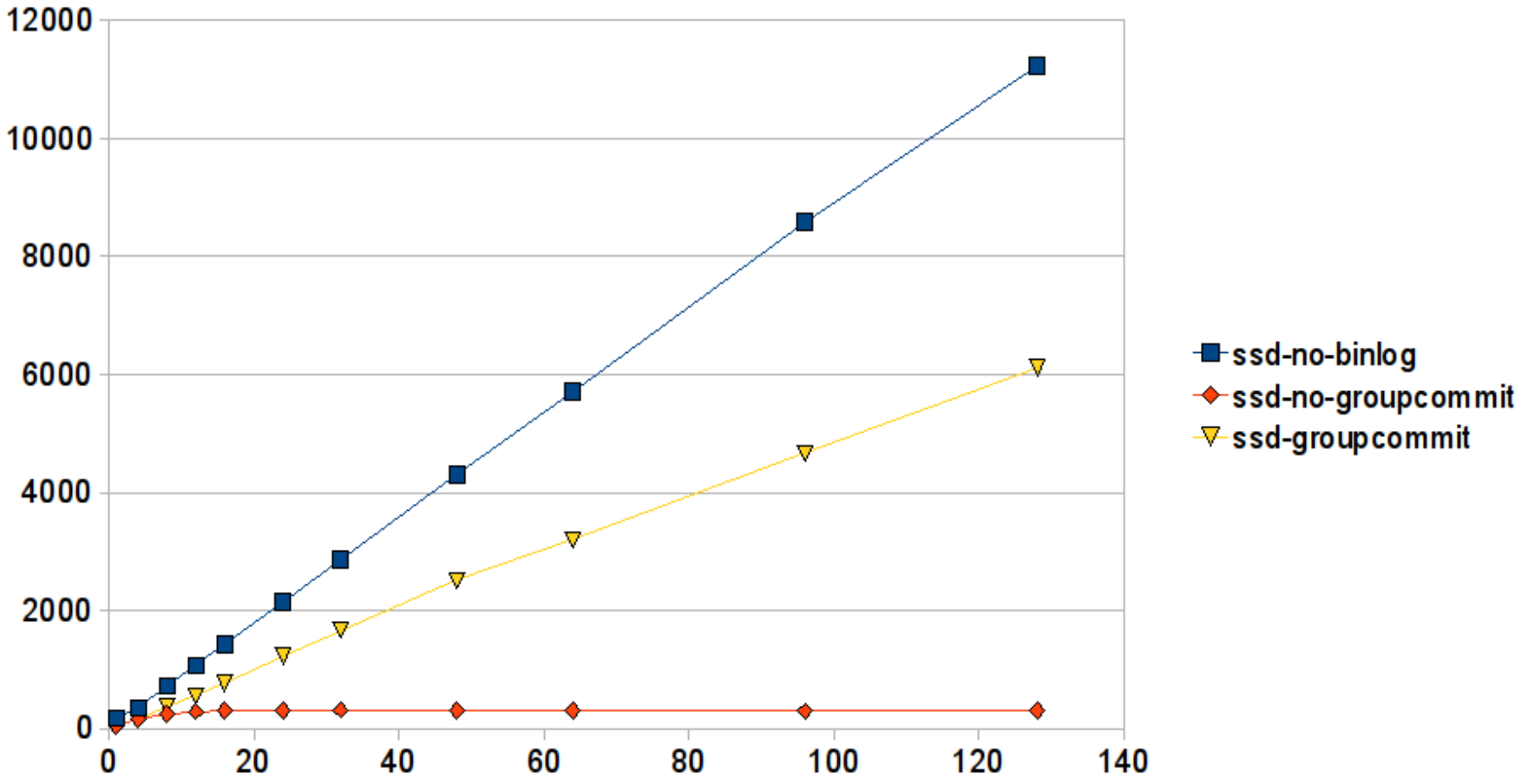
Commits per second vs. number of connections, RAID 1 HDD



- Yellow line shows group commit performance
- Now get scalability, only pay the cost of the 3 `*fsync()`

Group commit scales well

Commits per second vs. number of connections, RAID 1 SSD



- Yellow line shows group commit performance
- Now get scalability, only pay the cost of the 3 `*fsync()`

Extension to storage engine API

Extend the storage engine API

- `prepare()`
  - Write prepared trx in parallel, with group commit
- `prepare_ordered()`
  - Called serially, in commit order
- `commit_ordered()`
  - Called serially, in commit order
  - Fast commit to memory only
- `commit()`
  - Commit to disk in parallel, with group commit

Algorithm summary

- Storage engine API extension
  - Optional
  - Easy to implement for storage engine authors
- SMP-friendly behaviour
- Ensures consistent commit order between engine(s) and binlog
- Good performance improvement for parallel workloads
- Included in in MariaDB 5.3

Outline

Related work

Facebook group commit patch

- Group commit does not guarantee consistent commit order
  - User can disable group commit during backups
- Thread takes a ticket in InnoDB `prepare()`, waits for its turn in `commit()`

Mats Kindahl (Oracle) blog

- `http://mysqlmusings.blogspot.com/`
- Design sketch only, no published implementation
- Uses parallel `pwrite()` into binlog
- Seems not to handle the consistent commit order problem

START TRANSACTION WITH CONSISTENT SNAPSHOT

Consistent commit order allows to fix `START TRANSACTION WITH CONSISTENT SNAPSHOT`

- In MySQL, and in MariaDB $\leq$ 5.2, this does not do much
  - Suppose a transaction spans both InnoDB and PBXT
  - Can still happen that we see InnoDB part of a transaction, but not PBXT part
- With group commit this is fixed
  - Consistent snapshot sees all of a transaction, or nothing, also for multi-engine transactions

Avoiding FLUSH TABLES WITH READ LOCK

`START TRANSACTION WITH CONSISTENT SNAPSHOT` works for binlog too!

- New `binlog_snapshot_file` and `binlog_snapshot_position` status variables, similar to `SHOW MASTER STATUS`
- Obey consistent snapshot rules
- Can obtain master binlog position corresponding to given transaction snapshot
- Optimise `mysqlbinlog --single-transaction --master-data`
  - No more need for `FLUSH TABLES WITH READ LOCK`
  - Fully non-blocking slave provisioning
  - No stalling for long-running queries
  - XtraBackup still better for large/huge data sets

innodb_release_locks_early

- Re-write of the similar Facebook feature for the MariaDB group commit framework
- Optionally allows InnoDB to in-memory commit a transaction and release its row locks already during prepare phase
- Consistent commit order needed to make this safe for statement-based replication
- Can improve performance in the presense of hotspot rows
- Only “Mostly safe”, not full ACID, so off by default (Check MWL#163 or docs for details)

Future work: tunable sleep

- In group commit, if we deliberately sleep before writing to disk, more commits may arrive, reducing total number of `fsync()` calls needed
- But if no more arrive, will reduce performance
  - Can then be worse than without group commit
- No sleep implemented in first version currently
  - But status variables to monitor group commit performance
  - Easy to add sleep option later if experience shows it is needed
  - Eventually would be nice to have an optional auto-tuning sleep

Future work: further reducing `fsync()` calls

- We still need **three** `fsync()` calls, even if they can be shared among several commits.
- But suppose we omit the `fsync()` calls in InnoDB...
- Then at crash recovery, we may find transactions missing in InnoDB
  - But we can re-play them from the binlog!
- This idea again is enabled by having consistent commit order
  - Can start re-playing from a well-defined point
- Potential to further improve commit throughput by a factor of 3 (in the extreme case)
- This is MWL#164
  - `http://askmonty.org/worklog/Server-RawIdeaBin/?tid=164`

Future work: *even* further reducing `fsync()` calls

Another idea is to implement a group commit mode similar to `innodb_flush_log_at_trx_commit=2`

- No durability, but still consistent crash recovery
- No `fsync()` penalty at all (sync in background once per second).

Idea:

- Already after prepare phase, commit the transaction to memory and return to client
- Rest of commit algorithm happens in a background thread (`fsync()` calls in InnoDB and binlog)
- Same connection can even participate twice in the same group commit!
- Many applications need high commit throughput, and can sacrifice durability, but still need consistent crash recovery

Plea to MySQL@Oracle

- Let's avoid diverging storage engine APIs between MySQL@Oracle and the other variants
- Please start participating in the discussions
  - This work has been extensively documented and discussed already during the design phase
- Please stop ignoring all development outside of Oracle
  - Expecting everyone to sign SCA without anything in return is not reasonable

Outline

Conclusion

- Group commit available in MariaDB 5.3.
- The new framework enables several nice spin-off feature
- Big speedups possible in workloads with high transaction volume and high parallelism
- Much more affordable to run with crash-recovery enabled

Slides:

`http://knielsen-hq.org/maria/uc2011.pdf`

Contact:

`monty@askmonty.org`

`knielsen@askmonty.org`